Functional Programming and the Scala Language

Lecture 2 Continuation

Eugene Zouev
Innopolis University
Spring Semester 2018

To Remind:

- FP cornerstones: immutable objects & functions as values
- Scala: object-oriented meets functional; imperative and/or OO and/or functional paradigms
- Function definitions & local functions
- Functions & operators
- Function literals
- Closures

Today:

- Partially-applied functions
- Currying
- By-name parameters
- Tuples
- Traits

Scala: Functional objects (1)

Arrays, functional objects and apply function

```
val strings = new Array[String](3)
strings(0) = "Hello"
strings(1) = ", "
strings(2) = "world!\n"

for (i <- 0 to 2)
  print(strings(i))
```

Why access to array elements with parentheses, not brackets?

Scala: Functional objects (2)

Arrays, functional objects and `apply` function

Why access to array elements with parentheses, not brackets?

- Arrays are **classes** in Scala – as all other types.
- So, any concrete array is just an instance of that class.
- **All classes are "functional" in Scala: this means they have special `apply` function.**
- **A construct like `A(i)` is treated as a call to `apply`.**

Scala: Functional objects (3)

Arrays, functional objects and `apply` function

Compare with functional object definition in C++

So, the construct like

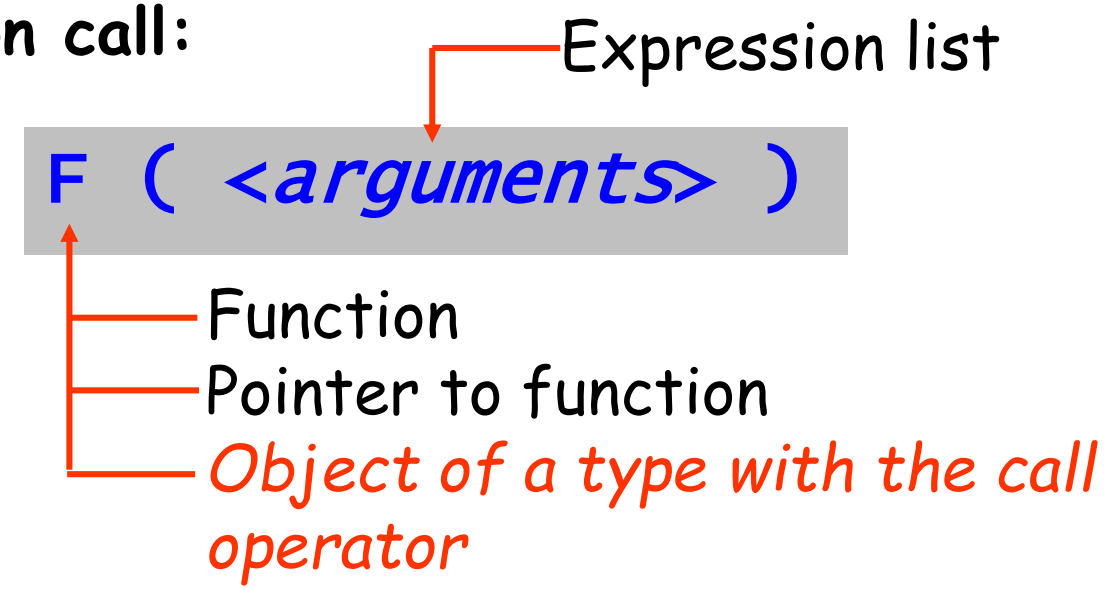
```
val element = strings(i)
```

is equivalent to

```
val element = strings.apply(i)
```

Functional Objects: a Side-step (1)

Function call:



Examples:

```
int F ( int x) { return expr; }
int (*pF)(int) = F;
 . . .
int a = F(1);
int b = pF(1);
```

```
class C {
public:
   int operator()(int x)
   { return expr; }
};
 . . .
C c;
int z = c(1); // ≡ c.operator()(1);
```

Functional Objects: C++ & Scala

C++

- If a type has the call operator `operator()` then the type is called **functional type**.

The simplest case: the C/C++ **pointer-to-function type** is *the functional type.*

Scala

- If a type has the `apply()` function defined then the type is called **functional type**.

If an object is of a functional type it is called **functional object**.

Scala: Functional objects (4)

Arrays, functional objects and `apply` function

Similarly, the construct like

```
strings(i) = "Hello"
```

is equivalent to

```
strings.update(i)
```

So, all operations on arrays are treated as calling corresponding functions

Scala: Functional objects (5)

Arrays creation: apply function again

```
val numNames: Array[String] = new Array[String](3)
numNames(0) = "one"
numNames(1) = "two"
numNames(2) = "three"
```

Usual ("old-fashioned") way of creating array instances

```
val numNames = new Array[String](3)
numNames(0) = "one"
numNames(1) = "two"
numNames(2) = "three"
```

A bit more advanced way for doing the same ("type inference")

```
val numNames = Array("one","two","three")
```

"True" functional way ☺

```
val numNames =
        Array[String]("one","two","three")
```

```
val numNames =
        Array.apply("one","two","three")
```

Factory method **apply**

Partially-applied functions (1)

When you invoke a function with arguments, you **apply** that function to the arguments. Example:

```
def sum(a: Int, b: Int, c: Int) = a + b + c
sum(1, 2, 3)  // returns 6
```

A **partially applied function** is an expression in which you **don't supply all** of the arguments needed by the function. Instead, you supply **some**, or **none**, of the needed arguments.

Partially-applied functions (2)

Continuing the example:

```
val a = sum(1, 2, 3) // function call

val b = sum(1, _: Int, 3)

  // b contains a function
  // with one integer parameter!

b(2)  // returns 6
b(5)  // returns 9
```

Scala: Currying (1)

A curried function is applied to **multiple argument lists**, instead of just one.

Example:

```
def trivialSum(x: Int, y: Int) = x + y
trivialSum(1,2)  // returns 3
```

Similar **curried** function:

```
def curriedSum(x: Int)(y: Int) = x + y
curriedSum(1)(2)  // returns 3
```

Scala: Currying (2)

What's happening when `curriedSum` is invoked?-
Actually, two traditional functions are invoked "back to back":
The first call takes one integer argument and returns functional value:

```
def first(x: Int) = (y: Int) => x + y
```

Applying this function to the value 1 gives the second function with one parameter:

```
val second = first(1)
```

Applying the functional object second to the value 2 gives the final result:

```
second(2)   // returns 3
```

Currying & Partially-applied Functions

```
def curriedSum(x: Int)(y: Int) = x + y
curriedSum(1)(2)  // returns 3
```

The function value:
the function with one par
returning function with one par

Placeholder for the second parameter list

```
val onePlus = curriedSum(1)_
onePlus(2)  // returns 3
```

"By-Name" Parameters (1)

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if ( assertionsEnabled && !Predicate() )
    throw new AssertionError
```

How to use:

```
myAssert(() => 5 > 3)
```

A bit awkward…

Why not to write just this:

```
myAssert(5 > 3)
```

Wrong!

"By-Name" Parameters (2)

```
var assertionsEnabled = true

def myAssert(predicate: => Boolean) =
  if ( assertionsEnabled && ! Predicate())
    throw new AssertionError
```

Now it's correct!

```
myAssert(5 > 3)
```

Scala Tuples (1)

Tuple is an arbitrary collection of several elements of different types – in contrast to arrays, lists and similar collections.

Tuples are used in any context where it's reasonable to make a single entity out of several ones. Perhaps the most typical example: to define a function returning several values.

```
val pair = (99, "ninety nine“)
```

Type of `pair` is deduced as `Tuple2[Int,String]`, where `Tuple2` is generic predefined class.

Scala Tuples (2)

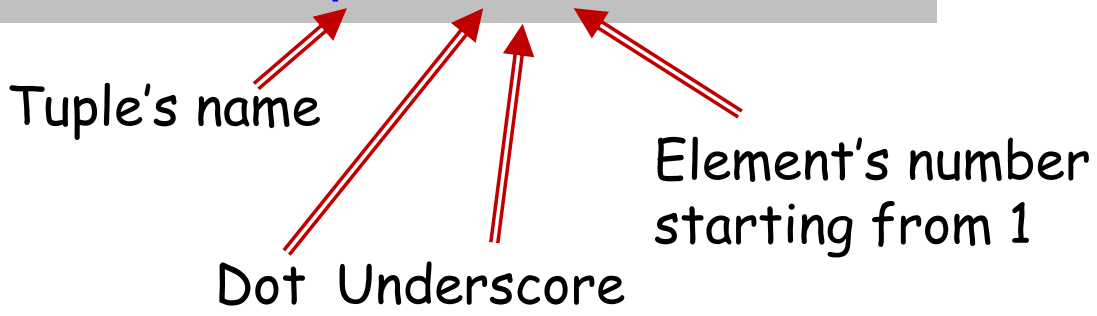
A typical example: function returning several values.

```
def QRoots(a, b, c: Double): (Double, Double)
{
   require(a != 0)
   require(b*b >= 4*a*c)

   val d = sqrt(b*b-4*a*c)
   val a2 = a*2
   ( (-b-d)/a2, (-b+d)/a2 )
}

val roots = QRoots(1,6,3)
```

Scala Tuples (3)

How to access to tuple's elements:

```
val pair = (99, "ninety nine“)

println(pair._1)
val second = pair._2
```



Why not use "familiar" notation like `pair(i)`?

- Tuples may contain elements of different types, therefore compiler cannot deduce type without exact knowledge about tuple elements' position…

Traits (1)

Two basic ways for establish **relationships** between classes/objects

- Inheritance: base & derived classes
- Interface: an abstraction of properties to be implemented by classes

Scala gives the third option:

- **Traits**.

Example

The task: to add *comparison functionality* to `Rational` class

Traits (2)

```
class Rational(n: Int, d: Int) {
  ...
  def <  (that: Rational) = (this.numer*that.denom)<(that.numer*this.denom)
  def >  (that: Rational) = that < this
  def <= (that: Rational) = (this<that) || (this==that)
  def >= (that: Rational) = (this>that) || (this==that)
  ...
}
```

Direct solution

Is it a good solution? – of course, yes! ☺

Three observations, however:

- Operators >, <= and >= are implemented using <
- These three operators will **be exactly the same** for any class providing comparison functionality.
- The specifics ("core semantics") of comparison for any class is solely within operator <

Traits (3)

How could solution using interfaces look like?

```
interface Ordered {
  ...
  def <  (that: Rational): Boolean
  def >  (that: Rational): Boolean
  def <= (that: Rational): Boolean
  def >= (that: Rational): Boolean
  ...
}
```

… and each class should provide its own implementation for each operator!

```
class Rational(n: Int, d: Int) : implements Ordered {
  ...
  def <  (that: Rational) = (this.numer*that.denom)<(that.numer*this.denom)
  def >  (that: Rational) = that < this
  def <= (that: Rational) = (this<that) || (this==that)
  def >= (that: Rational) = (this>that) || (this==that)
  ...
}
```

Traits (4)

Traits provide better solution

The trait is made generic to be used for any class

compare function encapsulates the core semantics of the comparison. It depends on the class whose objects are to be compared therefore it should be implemented in that class

```
trait Ordered[T] {
  def compare (that: T): Int  // returns 0, 1, or -1

  def <  (that: T): Boolean = (this compare that) < 0
  def >  (that: T): Boolean = (this compare that) > 0
  def <= (that: T): Boolean = (this compare that) <= 0
  def >= (that: T): Boolean = (this compare that) >= 0
}
```

The implementation of four comparison operators is here, and **it's the same** for each class that mixes in the trait

compare function is left undefined until the trait is used in a class ("mixed in" into the class).

Traits (5)

Using traits: "mix in" technology

Either **extends** or **with** keyword

Generic trait `Ordered` gets instantiated for `Rational` class

```
class Rational(n: Int, d: Int) extends Ordered[Rational]
{
  ...
  def compare (that: Rational) =
       (this.numer*that.denom) < (that.numer*this.denom)
  ...
}
```

`compare` function is the only thing the class should implement. It reflects the semantics of just *rational numbers*' comparison. All operators from the trait become applicable automatically.

The task for home thinking:
- What's the main difference between traits & interfaces??